HW1: Mid-term assignment report

Solomiia Koba [118313], v2025-11-03

1 Introduction

1.1 Overview of the work

This report presents the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy.
The project consists of a full-stack web application called *ZeroMonos Waste Management System*, designed to facilitate the booking and management of bulky waste collection services across multiple municipalities in Portugal. The system enables citizens to conveniently schedule collection appointments for items through a web portal, while providing municipal staff with tools to manage and track service requests.

1.2 Current implementation (faults & extras)

Limitations: CI/CD pipeline wasn't implemented.

A) Extra features: booking cancellations, lombook use, the "citizen facing" and "staff facing" services can be available from different pages (index.html and staff.html). Implemented a dashboard for the staff, enabling visual monitoring of requests and operations for each municipality. It's possible to sort by municipality and state. Date validation was added (it's not possible to book on Saturday and Sunday).

1.3 Use of generative AI

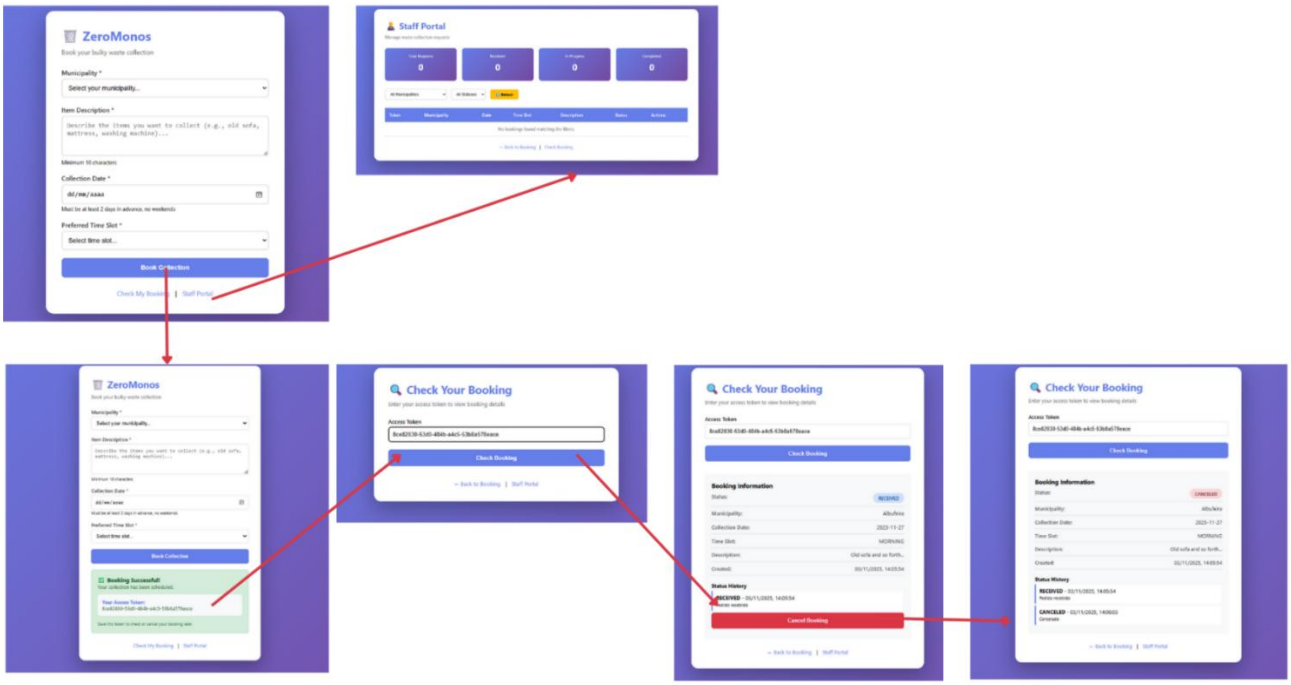
AI-assistant (Chat GPT) was used to generate user stories, initial HTML/CSS structure for web pages.

2 Product specification

2.1 Functional scope and supported interactions

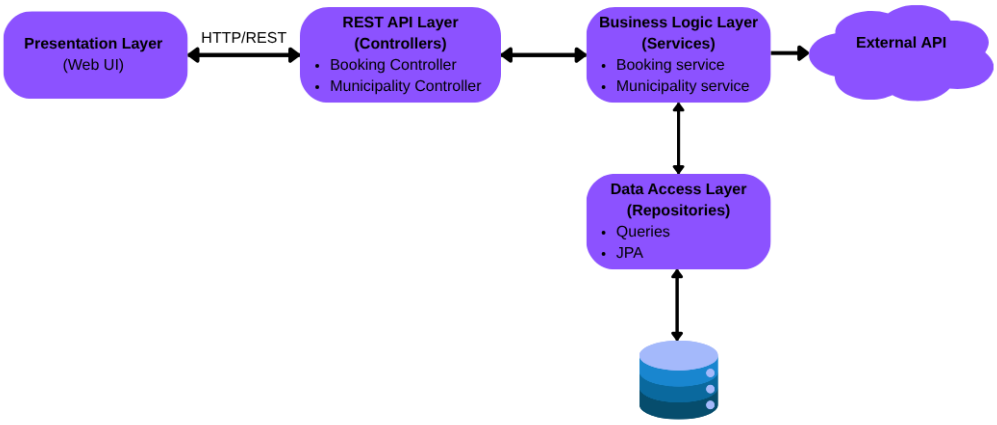
The application actors:

1. **Citizen**:
   - create booking, providing necessary information.
   - checks booking status entering generated token.
   - view history of the booking.
   - cancel active booking.
2. **Staff Member:**
   - view all requests.
   - change the state of each request (RECEIVED → ASSIGNED → IN_PROGRESS → COMPLETED).
   - filter requests by municipality and/or status.



2.2 System implementation architecture

The system follows a **layered architecture** pattern with clear separation of concerns.



Backend Framework: Spring Boot.
Database:

- PostgreSQL: Production database (persistent)
- H2 Database: Testing database (in-memory)

Build tool: Maven.
Java version: 19.
Frontend:

- HTML5: Semantic markup.
- CSS3: Styling with Flexbox/Grid.
- JavaScript ES6+: Client-side logic (Vanilla JS, no frameworks).

Testing Frameworks:

- JUnit 5: Unit testing framework.
- Mockito: Mocking framework for unit tests.
- Spring MockMvc: Controller integration testing.
- Selenium WebDriver: Browser automation.
- Cucumber: BDD testing framework.
- WebDriverManager: Automatic driver management.

Quality Assurance Tools:

- JaCoCo: Code coverage analysis.
- SonarQube: Static code analysis.
- Maven Surefire: Unit test execution.
- Maven Failsafe: Integration test execution.

Docker: used for running sonarqube and postgresql containers.

2.3 API for developers

Base URL: http://localhost:8080/api.

Endpoint	Method	Description
/api/bookings	POST	Create a new booking
/api/bookings/{token}	GET	Retrieve booking details
/api/bookings/{token}	DELETE	Cancel booking
/api/ /bookings	GET	List all bookings
/api/bookings/{token}/status	PUT	Update state

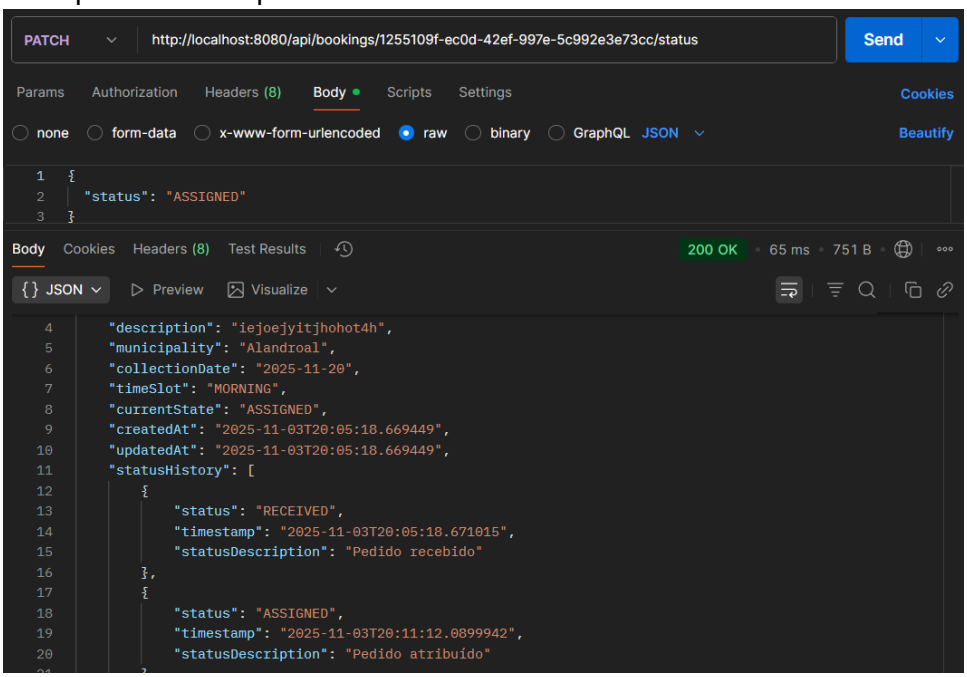
/api/municipalities	GET	List all municipalities

The following curl commands are provided as examples of how to interact with the API endpoints from the command line.

///// **Creating booking**

```
curl -X POST http://localhost:8080/api/bookings \
  -H "Content-Type: application/json" \
  -d '{
    "municipality": "Aveiro",
    "description": "Old washing machine and dryer",
    "requestDate": "2025-11-15",
    "timeSlot": "AFTERNOON"
  }'
```

Example of status update:



3 Quality assurance

3.1 Overall strategy for testing

I adopted a testing approach that combines test-after development for entities and repositories (straightforward CRUD), test-driven development (TDD) for business logic with complex rules, behaviour-driven development (BDD) for user-facing scenarios.

Testing level	Tools	Reason
Unit	JUnit 5 + Mockito	Standard Spring integration
Service with Mocks	Mockito	Isolate external dependencies
API Integration	MockMvc	
Functional	Selenium + Cucumber	BDD scenarios in Gherkin, browser automation

Performance	REST-Assured	Concurrent requests, response time measurement

3.2 Unit and integration testing

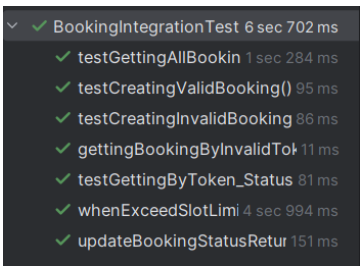
I used unit and integration tests service layer business logic in *BookingService* and *MunicipalityService*. I used Mockito for dependency mocking, inserted @ExtendWith for test setup and parameterized tests for multiple input scenarios. I simulated mocking of External API for fast testing and reliability and ensured that the system correctly validates input data, applies booking constraints, and interacts properly with the repository and external services. An example is shown below, where the test verifies that a valid booking request is processed correctly.

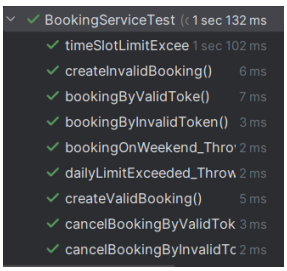
```
@Test  solomiiakoba
void createValidBooking() {
    when(municipalityService.isValidMunicipality("Aveiro")).thenReturn( t: true);
    when(repository.countActiveRequestsByDateAndMunicipality(any(), any())).thenReturn( t: 5L);
    when(repository.countActiveRequestsByDateAndMunicipalityAndTimeSlot(any(), any(), any())).thenReturn( t: 2L);
    when(repository.save(any(ServiceRequest.class))).thenReturn(serviceRequest);

    BookingResponseDTO response = bookingService.createBooking(validRequest);
    assertThat(response).isNotNull();
    assertThat(response.getToken()).isEqualTo( expected: "token");
    assertThat(response.getMunicipality()).isEqualTo( expected: "Aveiro");
    verify(repository, times( wantedNumberOfInvocations: 1)).save(any(ServiceRequest.class));
}
```

Integration testing was done with Spring MockMvc (Controller + Service + Repository) that uses h2 in-memory db to test the API endpoints as a whole, ensuring that the application context loads correctly and that all layers interact properly. The following example tests the /api/bookings endpoint, verifying that a valid booking request returns the expected response

```
@SpringBootTest  solomiiakoba
@AutoConfigureMockMvc
public class BookingIntegrationTest {
    @Autowired  11 usages
    private MockMvc mockMvc;
    @Autowired  8 usages
    private ObjectMapper objectMapper;
    @Autowired  1 usage
    private ServiceRequestRepository repository;

    @BeforeEach  solomiiakoba
    public void setup() {repository.deleteAll();}

    @Test  solomiiakoba
    void testCreatingValidBooking() throws Exception {
        BookingRequestDTO request = new BookingRequestDTO();
        request.setDescription("Old sofa and dryer...");
        request.setMunicipality("Aveiro");
        request.setCollectionDate(getNextWeekday());
        request.setTimeSlot(TimeSlot.MORNING);

        mockMvc.perform(post( uriTemplate: "/api/bookings")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isCreated())
                .andExpect(jsonPath( expression: "$.token").exists())
                .andExpect(jsonPath( expression: "$.municipality").value( expectedValue: "Aveiro"))
                .andExpect(jsonPath( expression: "$.currentState").value( expectedValue: "RECEIVED"))
                .andExpect(jsonPath( expression: "$.statusHistory").isArray())
```

Results of unit and integration tests:



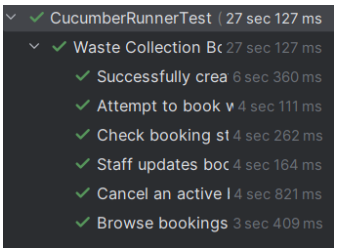


3.3 Acceptance testing

Using the approach Behavior-Driven Development (BDD) with Cucumber and Selenium, I wrote tests in natural language (Gherkin) for the client-side as well as the staff-side, focusing on user behavior. I implemented headless mode essential for CI/CD pipelines and WebDriverManager to eliminate driver management headaches.

```
@When("I select municipality {string}")  solomiiakoba
public void iSelectMunicipality(String municipality) {
    wait.until(ExpectedConditions.presenceOfElementLocated(
            By.id("municipality")));
    Select municipalitySelect = new Select(driver.findElement(
            By.id("municipality")));
    municipalitySelect.selectByVisibleText(municipality);
}
```

Results of the test:



3.4 Non-functional testing

Performance testing was implemented to simulate 10 concurrent users doing bookings on the system, api response time under load and to measure queries performance. The data confirms that the API can efficiently serve multiple front-end clients requesting location data with minimal latency.

========== GET MUNICIPALITIES PERFORMANCE ==========

Metric	Result
Number of requests bookings	50
Average response time	31.64ms

========== PERFORMANCE TEST RESULTS ==========

Metric	Result
Total concurrent users	10
Total execution time	492 ms
Average response time	489.2 ms

Metric	Result
Minimum response time	488 ms
Maximum response time	490 ms
Throughput	20.3 requests/second

========== DATABASE QUERY PERFORMANCE ==========

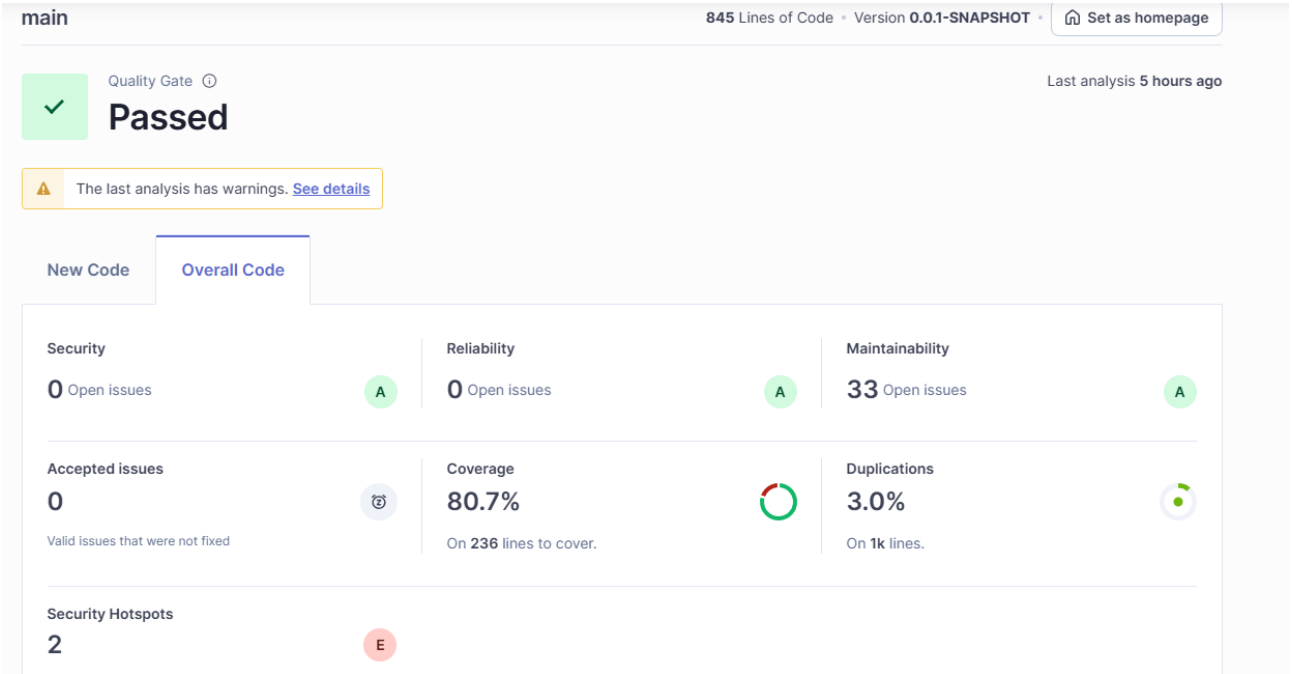
Metric	Result
Query time for all bookings	62 ms

3.5 Code quality analysis

Static code analysis was carried out using **SonarQube.** I was running SonarQube with (*while using H2*):

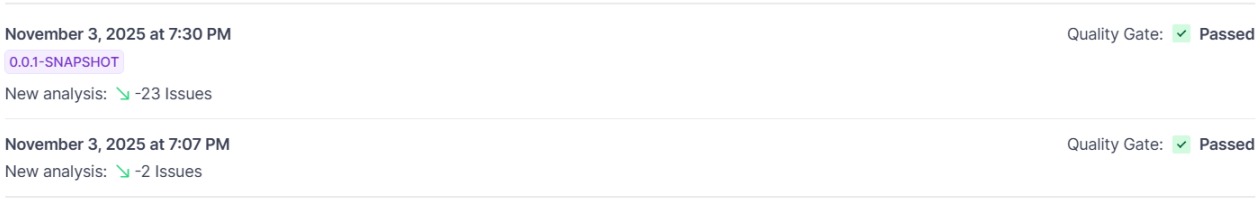
```
$ mvn clean verify sonar:sonar
        -Dsonar.projectKey=zeromonos
        -Dsonar.projectName='zeromonos'
        -Dsonar.host.url=http://localhost:9000  -
        Dsonar.token=sqp_496524a2e49f0b587b326eab1a50b7e3971d6230
```

The analysis shows that the project meets all Quality Gate conditions, with top scores **(A)** in security, reliability, and maintainability. The codebase has a solid **~81%** of test coverage, indicating that most business logic is validated through automated testing.
A small percentage **(3%)** of duplication was detected, mostly due to repeated DTO mapping. Maintainability warnings were related to minor code smells such as unused imports, overly long methods, and missing JavaDoc comments — issues that were easy to fix once highlighted by SonarQube.



Having corrected my code, I got 5 issues left.

I learned that it’s recommended to write without public and also that @CrossOrigin(origins = "*") allows any site to send requests to my backend. It’s safe in development but dangerous in production.



4 References & resources

Project resources

Resource:	URL/location:
Video demo	HW1/docs/demo-zeromonos.mp4

Reference materials

During the development of this project, several resources were consulted to better understand and implement the technologies involved – labs code, official documentation and the books recommended on the “cheat sheet” .

Cacheable (Spring Framework 6.2.12 API)
WebDriver | Selenium
Running Integration Tests :: Spring Boot
Maven Repository: org.springframework.boot
Bing Vídeos – full stack spring boot development